Lecture 9. Semantic Analysis – Scoping and Symbol Table

Wei Le

2015.10

Outline

- Semantic analysis
- Scoping
- The Role of Symbol Table
- Implementing a Symbol Table

Semantic Analysis

- Parser builds abstract syntax tree
- Now need to extract semantic information and check constraints
- Can sometimes be done during the parse, but often easier to organize as a separate pass
- Some things cannot be done on the fly during the parse, e.g., information about identifiers that are used before they are declared (fields, classes)
- Last phase in front end

Semantic Analysis

- Extract types and other information from the program
- Check language rules that go beyond the grammar
- Assign storage locations
- Key data structures: symbol tables – For each identifier in the program, record its attributes (kind, type, etc.)
- Find more errors

Semantic Analysis: Detecting Errors

- Lexical analysis: detecting illegal tokens (e.g., string constant too long, not legal characters in the string)
- Syntax analysis: structure errors, no parse tree for the string given grammar (brackets not match)
- Semantic analysis (dependent on the language): e.g., specify wrong types (int a $=$ b$+$ c (c cannot be string)), buffer overflows, divide by zero
    - scoping: e.g., all variables must be declared
    - typing: certain operations only can be performed on the certain types of variables; e.g., actual parameter types have to match formal parameter types

Examples of Semantic errors

- The type of the right-side expression of an assignment statement should match the type of the left-side, and the left-side needs to be a properly declared and assignable identifier (i.e. not some sort of constant).
- The parameters of a function should match the arguments of a function call in both number and type.
- The language may require that identifiers are unique, disallowing a global variable and function of the same name.
- The operands to multiplication operation will need to be of numeric type, perhaps even the exact same type depending on the strictness of the language.

Cool

1. All identifiers are declared
2. Types
3. Inheritance relationships
4. Classes defined only once
5. Methods in a class defined only once
6. Reserved identifiers are not misused

And others . . .

Scope

- Match identifiers' declaration with uses
- Visibility of an entity
- Binding between declaration and uses
- The **scope** of an identifier is the portion of a program in which that identifier is accessible
- The same identifier may refer to different things in different parts of the program: Different scopes for same name dont overlap
- An identifier may have restricted scope

Static vs Dynamic Scope

- Most languages have static scope: Scope depends only on the program text, not runtime behavior (e.g., COO, Java and C): most closely nested rule
- A few languages are dynamically scoped: Scope depends on execution of the program (e.g., Lisp, SNOBOL, Perl allow dynamic scope through certain keywords) – context specific scoping
- Dynamic scoping means that when a symbol is referenced, the compiler/interpreter will walk up the symbol-table stack to find the correct instance of the variable to use.

Static Scope

```
let x: Int <- 0 in
  {
      x;
      let x: Int <- 1 in
           x;
      x;
  }
```

Static Scope

```
let x: Int <- 0 in
  {
      x;
      let x: Int <- 1 in
          x;
      x;
  }
```

Uses of x refer to closest enclosing definition

Static scoping

```c
const int b = 5;
int foo()
{
   int a = b + 5;
   return a;
}

int bar()
{
   int b = 2;
   return foo();
}

int main()
{
   foo(); // returns 10
   bar(); // returns 10
   return 0;
}
```

Dynamic scoping

```c
const int b = 5;
int foo()
{
   int a = b + 5;
   return a;
}

int bar()
{
   int b = 2;
   return foo();
}

int main()
{
   foo(); // returns 10
   bar(); // returns 7
   return 0;
}
```

Scoping and Symbol Table

- Given a declaration of a name, is there already a declaration of the same name in the current scope, i.e., is it multiply declared?
- Given a use of a name, to which declaration does it correspond (using the "most closely nested" rule), or is it undeclared?
- Symbol table: a data structure that tracks the current bindings of identifiers (for performing semantic checks and generating code efficiently)
- large part of semantic analysis consists of tracking variable/function/type declarations .As we enter each new identifier in our symbol table, we need to record the type information of the declaration.

The Roles of Symbol Table

- Generally, symbol table is only needed to answer those two questions, i.e., once all declarations have been processed to build the symbol table, and all uses have been processed to link each ID node in the abstract-syntax tree with the corresponding symbol-table entry, then the symbol table itself is no longer needed, because no more lookups based on name will be performed (In practice, symbol table often keeps through the entire compilation and sometimes even beyond)
- The process of building symbol table, you are able to check the scoping rules

Implementing Symbol Table: Symbol Table Entry

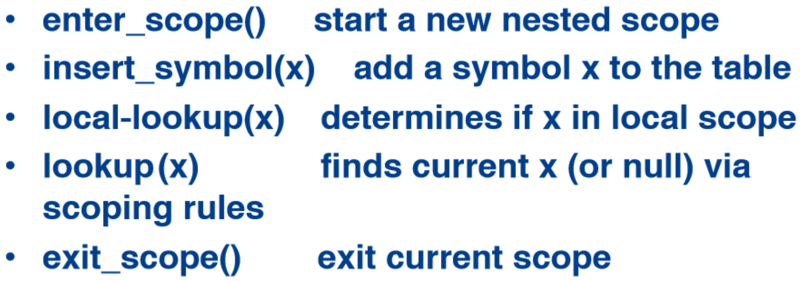
- Semantic of a variable: 1) When is used; 2) The effective context where a name is valid; 3) Where it is stored: storage address

Implementing Symbol Table: Key Operations

High level

- enter scope
- process a declaration
- process a use
- exit scope

low level



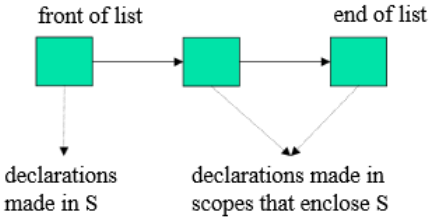
• enter_scope()	start a new nested scope
• insert_symbol(x)	add a symbol x to the table
• local-lookup(x)	determines if x in local scope
• lookup(x)	finds current x (or null) via scoping rules
• exit_scope()	exit current scope

Implementing Symbol Table: Data Structures

- **Unordered list: for a very small set of variables.**
- **Ordered linear list: insertion is expensive, but implementation is relatively easy.**
- **Binary search tree: $O(\log n)$ time per operation for n variables.**
- **Hash table: most commonly used, and very efficient provided the memory space is adequately larger than the number of variables.**

A List of Hash Table

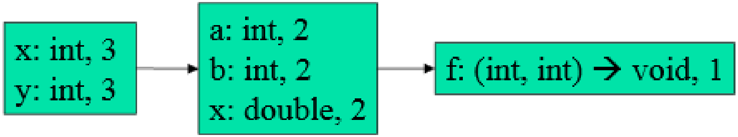
- One hash table for each currently visible scope
- At a given point, have symbol table accessible for each nesting level $\leq$ current nesting level



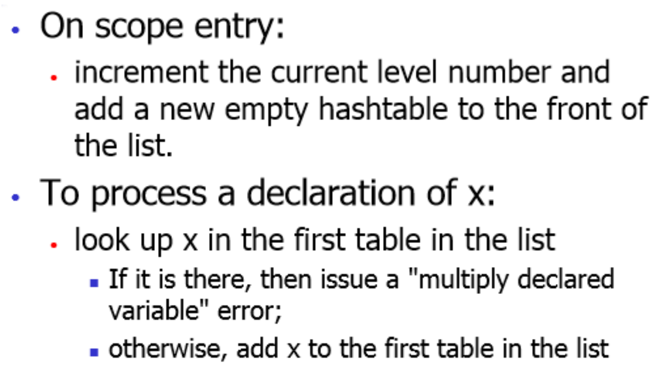
An Example

```
void f(int a, int b) {
   double x;
   while (...) { int x, y; ... }
}
void g() { f(); }
```

After processing declarations inside the while loop:

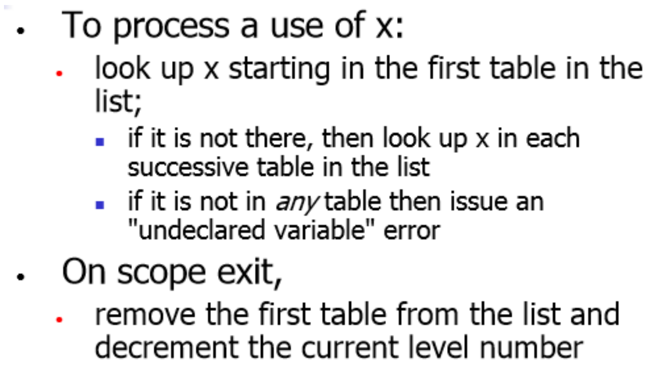


A List of Hash Table: Operations



- On scope entry:
  - increment the current level number and add a new empty hashtable to the front of the list.
- To process a declaration of x:
  - look up x in the first table in the list
    - If it is there, then issue a "multiply declared variable" error;
    - otherwise, add x to the first table in the list

A List of Hash Table: Operations



- To process a use of x:
  - look up x starting in the first table in the list;
    - if it is not there, then look up x in each successive table in the list
    - if it is not in *any* table then issue an "undeclared variable" error
- On scope exit,
  - remove the first table from the list and decrement the current level number

Insert Method Name

- Method names belong in the hashtable for the outermost scope
  - Not in the same table as the method's variables
- For example, in the previous example:
  - Method name f is in the symbol table for the outermost scope
  - Name f is *not* in the same scope as parameters a and b, and variable x
  - This is so that when the use of name f in method g is processed, the name is found in an enclosing scope's table

Runtime for Each Operation

1. **Scope entry**:
   - time to initialize a new, empty hashtable;
   - probably proportional to the size of the hashtable
2. **Process a declaration:**
   - using hashing, constant expected time (O(1))
3. **Process a use**:
   - using hashing to do the lookup in each table in the list, the worst-case time is O(depth of nesting), when every table in the list must be examined
4. **Scope exit**:
   - time to remove a table from the list, which should be O(1) if garbage collection is ignored

Scoping Example

- C++ does not use exactly the scoping rules that we have been assuming
  - In particular, C++ **does** allow a function to have both a parameter and a local variable with the same name
    - any uses of the name refer to the local variable
  - Consider the following code. What is the symbol table as it would be after processing the declarations in the body of *f* under:
    - the scoping rules we have been assuming
    - C++ scoping rules

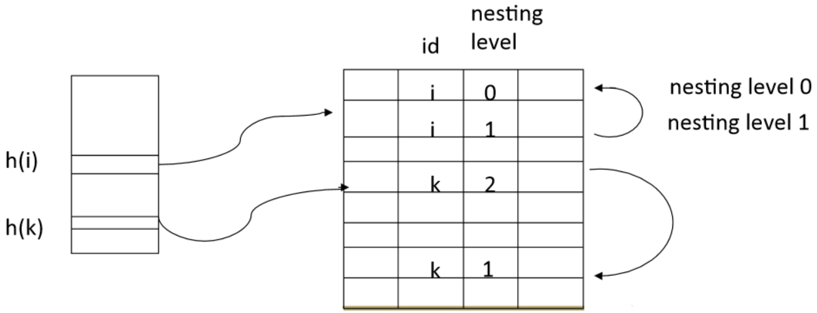
```
void g(int x, int a) { }
void f(int x, int y, int z) {
  int a, b, x; ...
}
```

Scoping Example

- Questions:
  - Which of the four operations described above
    - scope entry,
    - process a declaration,
    - process a use,
    - scope exit
  - would we need to change to use the following rules for name reuse instead of C++ rules:
    - the same name can be used within one scope as long as the uses are for different kinds of names, and
    - the same name *cannot* be used for more than one variable declaration in a nested scope

Single Hash Table to Implement Symbol Table

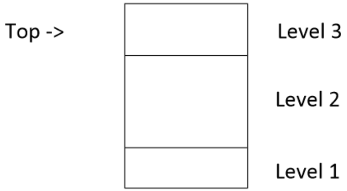
- Link together different entries for the same identifier and associate nesting level with each occurrence of same name.
- The first one is the latest occurrence of the name, i.e., highest nesting level



Single Hash Table to Implement Symbol Table

- During exit from a procedure - delete all entries of the nesting level we are exiting

1. Search for the correct items to remove – rehash -- expensive
2. Use extra pointer in each element to link all items of the same scope (scope chain)
3. Use an active ST stack that has entries for each scope

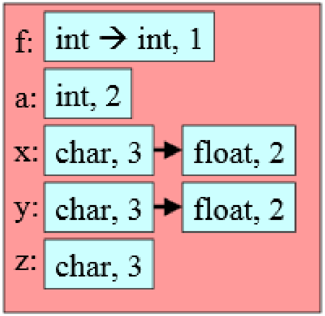


Pop entry and delete from hash table - keep stack entries around

An Example

```
int f (int a) {
  float x, y;
  while (...) {
    char x, y, z;
  }
}
void g () {
  int x;
  f(1);
}
```

- After processing the declarations inside the while loop:



A List of Operations

- On scope entry:
  - increment the current level number
- To process a declaration of x:
  - look up x in the symbol table
    - If x is there, fetch the level number from the first list item.
      - If that level number = the current level then issue a "multiply declared variable" error;
      - otherwise, add a new item to the front of the list with the appropriate type and the current level number

A List of Operations

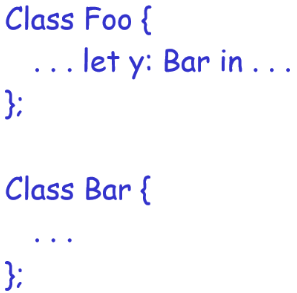
- To process a use of x:
  - look up x in the symbol table
  - If it is not there, then issue an "undeclared variable" error
- On scope exit:
  - scan all entries in the symbol table, looking at the first item on each list
  - If that item's level number = the current level number, then remove it from its list (and if the list becomes empty, remove the entire symbol-table entry)
  - Finally, decrement the current level number

Scope Rules for Cool

Beyond most closely nested rule:

- Class definition: globally visible, a class can be used before it is defined

An Example:



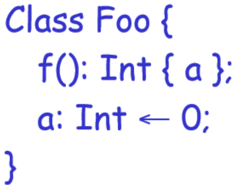
```
Class Foo {
   . . . let y: Bar in . . .
};

Class Bar {
   . . .
};
```

Scope Rules for Cool

- Class names can be used before being defined
  - We can't check this property using a symbol table
  - Or even in one pass
- Solution:
  - Pass 1: Gather all class names
  - Pass 2: Do the checking
- Semantic analysis requires multiple passes: probably more than two

Scope Rule for Cool

Attribute names are global within the class in which they are defined
An example:



```
Class Foo {
   f(): Int { a };
   a: Int ← 0;
}
```

Scope Rule for Cool

- Method and attribute names have complex rules
- A method need not be defined in the class in which it is used, but in some parent class
- Methods may also be redefined (overridden)

Implementing a Symbol Table for Object-Oriented Programs

- Single global table to map class names to class symbol tables
  - Created in pass over class definitions
  - Used in remaining parts of compiler to check field/method names and extract information
- All global tables persist throughout the compilation (and beyond in a real Java or C compiler...)
- One or multiple symbol tables for each class: 1 entry for each method/field; Contents: type information, public/private, storage locations (later)
- Local symbol table for each method 1 entry for each local variable or parameter Contents: type information, storage locations (later)
- the scope for a derived class is often placed inside of the scope of a base class

Summary

- Concepts: Scope (static and dynamic scope), Symbol Table
- Basic scoping rules (different programming languages can implement different rules):
  - variables have to be declared before use
  - variables should be declared twice
  - most closely nested rule: the use of a variable always binds to the declaration that is in the closet scope
- Implementation of a symbol table
  - Symbol table entry: type, storage type, scope information
  - Data structure
  - Operations
  - Complexity
  - Comparisons of a set of algorithms in terms tradeoffs